Decomposing Correctness Proofs of Fault-Tolerant Algorithms[*]

Lee Pike

Formal Methods Group
NASA Langley Research Center
M/S 130, Hampton, VA 23681-2199
USA
`lee.s.pike@nasa.gov`

Summary. Proving the correctness of fault-tolerant algorithms is a tedious endeavor. Not only are the algorithms themselves complex, but nondeterministic fault transitions due to the environment compound the complexity. We demonstrate how to systematically decompose proofs of correctness. We first define a relation between the execution of an algorithm in a fault-generating environment and a fault-free one. Proofs can then be decomposed as follows. First, an algorithm is shown to satisfy this relation. Second, proving that the algorithm satisfies its requirements necessitates reasoning about the execution of the algorithm in a fault-free environment only; faulty behavior can be ignored. This technique systematizes and simplifies proofs of fault-tolerant algorithms. The verification of the Oral Messages(1) algorithm is given as a simple illustrative example.

1 Introduction

Fault-tolerant embedded systems are rapidly being integrated into the digital control systems of automotive, aircraft, and other such systems [11, 12, 27]. In such contexts, these systems are safety-critical (i.e., lives may be lost if they fail), so it is imperative that they operate correctly. Thus, the algorithms these systems employ *must* be correct. This is often difficult to achieve given that fault-tolerant algorithms can be notoriously tedious and complex. Not only must a fault-tolerant algorithm perform an intended function, but it must mask faults.[2] Furthermore, fault-tolerant algorithms often operate in a distributed environment in which issues such as communication and synchrony must be addressed.

Proofs purporting to demonstrate the correctness of fault-tolerant algorithms are likewise complex. Hand proofs – even those appearing in peer-reviewed journals – of the correctness of relatively simple fault-tolerant algorithms have been flawed [17]. Formal verification efforts, although arguably less error-prone, are arduous tasks due to these complexity issues. For example, see [8, 15, 17, 23].

---

[*] Submitted to *Formal Techniques in Fault-Tolerant and Real-Time Systems* (FTRTFT), 2004.

[2] In fact, the fault-masking itself has been found to be the principle sources of errors in some systems [25].

The formal methods and fault-tolerance communities are consequently motivated to systematize and simplify these proof-of-correctness efforts [26, 14]. We pursue this goal by presenting a general method to decompose these proofs. This decomposition rests on the formal definition of fault-tolerant correctness in Sect. 5.1. The essential idea is this: first, we show that a fault-tolerant algorithm executing in a fault-generating environment is equivalent, in a certain sense, to its execution in a fault-free environment. If this relation holds, we can prove properties of the algorithm hold for executions in fault-generating environments based on a model of its execution in a fault-free environment.

Decomposing the verification effort into these two steps reduces the complexity of the proof. A primary complication in reasoning about fault-tolerant algorithms is that faults introduce nondeterminism into the algorithm execution. Our goal is to reduce this complication by shifting much of the proof burden to reasoning about fault-tolerant algorithms in fault-free environments. With the method described, faults need to be explicitly reasoned about only once. Thereafter, in proofs demonstrating an algorithm satisfies its requirements, they can be ignored. Finally, decomposed proofs are more systematic – proofs of correctness for various algorithms can be accomplished using the same proof technique.

2 Organization

In Sect. 3, we introduce problem constraints and present basic definitions. We then examine some related work in Sect. 4. Section 5 defines the central concept of *fault-tolerant correctness*, and it explains how to model the execution of an algorithm in fault-generating and fault-free environments. Section 6 applies the techniques described to the familiar Oral Messages(1) algorithm [13, 21]. Section 7 contains concluding remarks and proposed future work.

3 Constraints and Definitions

We begin by constraining the problem. We are interested in distributed deterministic fault-tolerant algorithms. Our model of a distributed system follows those in [24, 18]. Notably, we assume each process has a unique and globally-known process identifier. In the nomenclature of [18], we consider fault-tolerant algorithms in synchronous systems[3] that handle process failures (however, we explain how to model link failures as a kind of process failure in Sect. 6). We consider Byzantine faults [13, 7], the most severe and general kind of fault, in this initial investigation. Nevertheless, these results are extensible to a more refined fault model.

Our basic definitions are as follows.

---

[3] The results herein do not depend on synchrony, but basic fault-tolerance e.g., reaching agreement, is often impossible in an asynchronous setting [18].

Definition 1 (Design Specification). *The design specification specifies how an algorithm behaves, at some level of abstraction.*

State machines [16], petri nets [22], and recursive functions [2] (the emphasized model in this paper) are some common frameworks for expressing an algorithm's design specification.

Definition 2 (Requirement). *A requirement is a post-condition that should hold after the algorithm terminates, provided certain pre-conditions hold. For distributed algorithms, a requirement is a relation between tuples of non-faulty process inputs and outputs. (If a requirement needs no pre-conditions to hold, then it simply relates all possible inputs to the prescribed outputs of non-faulty processes.) More formally, let* I *and* O *be* n*-tuples of inputs and outputs, respectively. In a tuple, we denote the value at index* j *with a subscript, so* $v_j \in I$ *denotes the input of process* j*. Denote requirement* R *holding of these tuples by* $R(I, O)$*.*

We stipulate that requirements are preserved under requirement restriction.

Definition 3 (Requirement Restriction). *If* $R(I, O)$*,* N *is a nonempty subset of the indicies of these tuples, and* I' *and* O' *are tuples created from* I *and* O*, respectively, by taking each* v_j *such that* $j \in N$*, then* $R(I', O')$*.*

This is a common property for a requirement to have; for example, agreement and validity [18] conditions (instances of which are given in Sect. 6.1) satisfy requirement restriction.

Definition 4 (Requirements Specification). *A requirements specification provides the set of requirements that an algorithm must satisfy.*

Definition 5 (Correctness). *Proving an algorithm* correct *requires demonstrating that its design specification satisfies each requirement in the requirements specification.*

Design specifications in general make implicit assumptions about the environment in which the specified algorithm executes. For example, inherent in design specifications for fault-intolerant algorithms is the assumption that the environment does not generate faults in actions like sending/receiving messages and reading/writing data. However, because fault-tolerant algorithms are meant to mask such faults generated by the environment, their specifications must not a priori assume the environment is completely fault-free. We therefore distinguish design specifications that make this assumption from those that do not.

Definition 6 (Fault-Free Specification). *The fault-free specification of a fault-tolerant algorithm is the design specification in which the environment is assumed to be fault-free.*

Definition 7 (Fault-Tolerant Specification). *The fault-tolerant specification is the design specification in which the environment is allowed to be fault-generating.*

Of course, no fault-tolerant algorithm tolerates all possible faults generated by the environment; a maximum fault assumption constrains the faults tolerated.[4]

---

[4] We do not consider faults arising from design errors.

Definition 8 (Maximum Fault Assumption (MFA)). *A specification of the number of and kinds of faults that an algorithm is designed to tolerate.*

In this investigation, the only kind of faults we consider are Byzantine faults, so the MFA will simply restrict their number. When verifying the correctness of the algorithm, we consider only those fault-generating environments satisfying the MFA.

4 Related Work

Proofs of correctness for fault-tolerant algorithms have generally proceeded by proving that an algorithm's fault-tolerant specification satisfies its requirements specification directly. Nevertheless, we know of a few endeavors related to the technique we present.

In [31], Weber uses a state machine approach to formally define fault-tolerance and to prove the correctness of fault-tolerant systems. He states that a system is fault-tolerant "if its *behavior in the presence of faults is the same as it would have been in the absence of faults*," [31]. Unfortunately, his definition applies only to systems that completely mask faults (the worked example he gives is of a storage management system with a redundant disk). Most fault-tolerant systems (or algorithms) do not completely mask faults (these issues are discussed in Sect. 5.1). In this respect, the work here can be seen as a significant generalization of Weber's approach.

In the Reliable Computing Platform (RCP) project at the NASA Langley Research Center, faults are masked completely, in accordance with Weber's definition of fault-tolerance. The goal of the RCP project is to formally specify and verify a synchronized fault-tolerant operating system [30]. The proof method employed in the RCP project is to develop a hierarchy of specification abstractions for the RCP and then to show a satisfaction relation holds between the various levels. The top level is the Uniprocessor System layer (US) specifying the behavior of RCP as if it were running on a single fault-free processor. The next highest specification level is the Replicated Synchronous layer (RS) in which the system is specified as a synchronous fault-tolerant system running on replicated processors with majority voting.

Others have made similar distinctions between behavior and fault-tolerance. In [1], a theory is proposed demonstrating that fault-tolerance is the addition of certain fault *detectors* and *correctors* to a fault-intolerant algorithm. The fault-tolerant specification of an algorithm is essentially a particular composition of the fault-generating environment, the fault-intolerant algorithm, and the detectors and correctors. Chandy and Misera take a similar compositional approach using their UNITY programming language to specify and verify fault-tolerant algorithms [5].

Finally, in [14], Lamport argues that reasoning compositionally about distributed systems is counterproductive. The sort of composition described therein is the composition of parallel components in the system.

> Any proof in mathematics is compositional – a hierarchical decomposition of the desired result into simpler subgoals. . . . Mathematics provides more general and more powerful ways of decomposing a proof than just writing a specification as the parallel composition of separate components [14].

We do not propose to decompose proofs with respect to parallel computation. His critique is orthogonal to the sort of proof decomposition we propose; in fact, these remarks seem to support the sort of decomposition we propose, as it is a "mathematical decomposition."

5 Decomposing Proofs of Correctness

We describe how to decompose the specifications of correctness of fault-tolerant algorithms in this section. In Sect. 5.1, we first define a relation between fault-tolerant algorithms executing in fault-generating and fault-free environments, respectively. In Sect. 5.2, we demonstrate how to efficiently model the execution of fault-tolerant algorithms as the evaluation of recursive functions to facilitate proving the relation holds. The results in this section are applied in Sect. 6.

5.1 Defining a Fault-Tolerance Relation

Fault-tolerance is canonically defined as correct behavior in the presence of (a specified kind and number of) faults [6, 28]. It would be convenient if correct behavior simply meant that faults are masked or corrected, and the output of the algorithm is the same regardless of whether faults occur. Then an algorithm would be fault-tolerant if for given inputs, its output in a fault-generating environment is equivalent to its output in a fault-free environment.

Unfortunately, this level of fault-tolerance is often impossible to achieve. Therefore, correct behavior is often defined in terms of agreement, validity, and termination conditions [18] (the last of which is usually trivial). For typical algorithms satisfying these requirements, it is easy to see that while agreement can be reached in the presence of faults, processes may reach agreement on different values in fault-generating and fault-free environments for the same inputs (we leave this as an exercise).

To prove a fault-tolerant algorithm satisfies agreement, validity, or any other requirement usually requires reasoning about the execution of the algorithm in the face of faults that nondeterministically alter its execution. Such proofs can be notoriously complex. This is the case even when requirements do not essentially relate to the fault-tolerance capabilities of the algorithm. What's more, such reasoning about fault transitions must be undertaken for each requirement to be satisfied.

We are therefore motivated to reduce this complexity by decomposing the proof process. This decomposition hinges on Def. 9, stipulating a relation between an algorithm executing in a fault-generating environment and a fault-free environment. Proofs can then be decomposed as follows. First, an algorithm is shown to satisfy this relation. Second, if the relation is satisfied, proving that the algorithm satisfies its requirements requires reasoning about the execution of the algorithm in a fault-free environment only; faulty behavior can be ignored.

To define this relation, we begin by noting that the execution of a fault-tolerant algorithm in a fault-free environment may approximate its execution in a fault-generating one in the following sense. In a fault-generating environment, suppose each process has some input. If there is some new assignment of inputs to the faulty processes (while the non-faulty processes have the same inputs) such that the outputs of the non-faulty processes are the same in both environments, the executions in the different environments are indistinguishable.

In the following definition, let A be a distributed fault-tolerant algorithm, and MFA its maximum fault assumption. Let $\mathit{ENV}^{\mathrm{fg}}$ be any fault-generating environment satisfying MFA.

Definition 9 (Fault-Tolerant Correctness). *Let each process i have input x_i, and let the output of each process i be z_i after the execution of A in $\mathit{ENV}^{\mathrm{fg}}$. Let N be the set of non-faulty processes in $\mathit{ENV}^{\mathrm{fg}}$. Then for each process i, there exists an input y_i such that for all $j \in N$, $y_j = x_j$ and each process $j \in N$ has output z_j after the execution of A in a fault-free environment.*

For all processes not in N, y_i can be any value that ensures processes in N have the correct outputs. Likewise, their outputs can be any value. The value y_i is unconstrained since we assume all faults are Byzantine faults. Under a more refined fault model, the inputs of these processes would be constrained according to MFA.

The following theorem assures that requirements that hold of the executions of an algorithm in a fault-free environment imply they hold of executions in a fault-generating environment, recalling Defs. 2 and 3.

Theorem 1. *Let A be a distributed fault-tolerant algorithm that is fault-tolerant correct. Suppose that for all input-tuples I, the execution of A in a fault-free environment generates an output-tuple O such that for requirement R, $R(I, O)$ holds. Then for all inputs, and all fault-generating environments $\mathit{ENV}^{\mathrm{fg}}$ that satisfy the maximum fault assumption for A, the execution of A in $\mathit{ENV}^{\mathrm{fg}}$ also guarantees $R(I', O')$ to hold, where I' and O' are the respective restrictions to non-faulty processes.*

Proof. If A is fault-tolerant correct, the output of a non-faulty process after the execution of A in $\mathit{ENV}^{\mathrm{fg}}$ is its output after A executes in a fault-free environment, where only the inputs to faulty processes differ. Since the inputs and outputs of non-faulty processes in the two environments are the same, $R(I', O')$ is a restriction of $R(I, O)$. □

5.2 The Design Specification

Recursive functions are often preferred when providing high-level behavioral specifications of synchronous fault-tolerant algorithms, such as in [26, 17], and especially when using theorem provers (e.g., PVS [20] and ACL2 [32]). Our concern here is to efficiently specify algorithms executing in fault-generating and fault-free environments as recursive functions.

We model the execution of a fault-tolerant algorithm A in a fault-free environment first. To model the inputs, we let $\lambda i.\, x_i$ be a function from the set of processes on which A executes to the input x_i of each process i. Since no faults are generated in a fault-free environment, all steps that depend on the environment are deterministic. We can thus specify A with a recursive function $\mathcal{A}^{\mathrm{ff}}$, where "ff" denotes *fault-free*.

Let the body of $\mathcal{A}^{\mathrm{ff}}$ contain functions $\overrightarrow{\mathit{env}}^{\mathrm{ff}} = \mathit{env}^{\mathrm{ff}}_0, \mathit{env}^{\mathrm{ff}}_1, \ldots, \mathit{env}^{\mathrm{ff}}_m$ that model the deterministic transitions of the environment (the functions in $\overrightarrow{\mathit{env}}^{\mathrm{ff}}$ may have differing signatures). The functions $\overrightarrow{env}^{\mathrm{ff}}$ model the actions that depend on the environment insofar as a faulty process could disrupt them. For example, these functions could model the sending of messages over a network or the writing of messages to a disk. In a fault-free environment, the message to be sent or written are faithfully sent and written, so these functions will often just be identity maps. For instance, let STR be a set of bit strings, and $write : STR \to STR$ a function that takes a bit string, and models the action of writing it to disk. The output is the bit string written to disk. In a fault-free environment, $write(str) = str$ since the string to be written is exactly the same as the string written.

In providing the fault-tolerant specification of A, we must extend the behavior allowed by $\overrightarrow{env}^{\mathrm{ff}}$ so that both intended and unintended behavior are possible. We formalize this using *uninterpreted functions* [19].

Definition 10 (Uninterpreted Function). *An uninterpreted function* $f : D \to R$ *has no defining body. For* $d \in D$, $f(d) \in R$ *is an uninterpreted constant.*

An uninterpreted function models Byzantine behavior by allowing for all possible behaviors [26]. The example in Sect. 6 demonstrates this usage. For a more refined fault model, such functions can be partially-interpreted.

Let the recursive function $\mathcal{A}^{\mathrm{ft}}$ ("ft" denotes *fault-tolerant*) be the fault-tolerant specification of A, such that its body contains only partially interpreted environmental functions $\overrightarrow{env}^{\mathrm{fg}}$. As noted, $\mathcal{A}^{\mathrm{ff}}$ is the fault-free specification of A. The bodies of $\mathcal{A}^{\mathrm{ft}}$ and $\mathcal{A}^{\mathrm{ff}}$ differ only by the terms $\overrightarrow{env}^{\mathrm{fg}}$ and $\overrightarrow{env}^{\mathrm{ff}}$.

To prove $\mathcal{A}^{\mathrm{ft}}(\lambda i.\, x_i)$ satisfies fault-tolerant correctness, we show that there exists an input function $\lambda i.\, y_i$ such that $\mathcal{A}^{\mathrm{ft}}(\lambda i.\, x_i) = \mathcal{A}^{\mathrm{ff}}(\lambda i.\, y_i)$[5] and that satisfies the constraints of Def. 9. Provided this relation holds, we can substitute $\mathcal{A}^{\mathrm{ff}}$ for $\mathcal{A}^{\mathrm{ft}}$ in proofs to show that an algorithm satisfies its requirements in the presence of faults.

6 An Extended Example: OM_1

We verify the fault-tolerant correctness of the Oral Messages(1) algorithm (OM_1) [13, 21, 17], and we use this result to prove it satisfies its requirements, agreement and validity. OM_1 is a simple and well-known fault-tolerant algorithm used to pass messages in a distributed network from a fixed single process (called the "general") in the network. The OM_1 algorithm proceeds in two rounds of communication with a computation step at the end of the second round (we assume a synchronous model of computation where time is abstracted to discrete rounds [18]). The algorithm is as follows.

Round 0
- The general sends its message to every process in the network.

Round 1
- Each process sends the message it received in Round 0 to every other process in the network. (If the sender fails-silent and no message was

---

[5] For modeling efficiency, we allow $\mathcal{A}^{\mathrm{ff}}$ and $\mathcal{A}^{\mathrm{fg}}$ to take axillary inputs, e.g., a program counter (see Sect. 6 for an example).

received in round 0, then a dummy value is sent.) Each process computes the majority message of those received in Round 1. If there is no majority, a special constant (e.g., no_maj) is computed.

6.1 The Requirements Specification and the MFA

The following two properties together are the requirements specification for OM_1:

Definition 11 (OM_1 Agreement). *Any two non-faulty processes compute the same value.*

Definition 12 (OM_1 Validity). *Assuming the general is non-faulty, each non-faulty process computes the value sent by the general.*

For OM_1 to satisfy these properties, we must constrain the faults it encounters. We specify the kinds of faults and the maximum number of them in the MFA.

We allow any kind of process fault to occur. A process may stop computing, it may compute values incorrectly, it may stop sending or receiving messages, or it may send incorrect messages. We allow processes to exhibit Byzantine behavior [13, 7] as well; i.e., a process may send different values to different receiving processes. Some of these behaviors may be the result of faulty channels rather than faulty processes. We abstract away this difference and say that process j is faulty if either process j is faulty or a communication channel sending messages from j to other processes is faulty [24]. This allows us extend the model of process faults to cover link faults, too. The MFA for the algorithm OM_1 follows.

Definition 13 (MFA for OM_1). *There are at least four processes, and no more than one of them is faulty.*

6.2 Specifying OM_1

Here we formally specify the execution of OM_1 in fault-generating and fault-free environments. The general approach in this section is similar to Rushby's model of the same algorithm in [26].

We let the set of messages sent and received by processes be modeled as a set MSG. I is a set of process indicies, and let variables i, j, k, and l range over I. We use the variable $g \in I$ when we particularly want to denote the general. VAL is a set of *process functions*, such that for $val \in VAL$, $val : I \rightarrow MSG$ takes a process index i and returns the message of process i. VAL provides a global view of what messages each process contains at some time. We say that "$val(i)$ is the value of process i," or "process i contains message $val(i)$."

Modeling Faults and the Environment

Here, we introduce the transitions dependent upon the environment. In a fault-free environment, processes are never faulty and so always send messages correctly. To model non-faulty communication from process j to process k, we introduce a function $send^{\mathrm{ff}} : I \times I \times VAL \rightarrow MSG$ that takes as arguments the indicies of sending and receiving processes, j and k, respectively, and a process function, and it returns the message received by process k from process j (again, 'ff' denotes *fault-free*). In a fault-free environment, the value sent by process j is just the value process j contains, so we let

$$send^{\mathrm{ff}}(j, k, val) \overset{\mathrm{df}}{=} val(j) .$$

As for modeling fault-generating environments, we model all process faults in terms of the messages a process sends. Abstractly, we allow all processes to receive and compute messages correctly, but faulty processes (including processes sending over faulty communication channels) may not communicate these messages correctly [24].

In a fault-generating environment, we take it that some processes are good (i.e., non-faulty) while others are faulty.[6] Given our abstractions, if a process is good, then all communication channels coming from the process are good, too. We introduce a predicate *good*? to distinguish these two sets of processes. We thus define the function $\mathit{send}^{\mathrm{fg}} : I \times I \times \mathit{VAL} \rightarrow \mathit{MSG}$ as

$$\begin{aligned} &\mathit{send}^{\mathrm{fg}}(j,k,\mathit{val}) \stackrel{\mathrm{df}}{=} \\ &\quad \text{if } \mathit{good?}(j) \text{ then } \mathit{send}^{\mathrm{ff}}(j,k,\mathit{val}) \\ &\quad \text{else } \mathit{bad\_send}^{\mathrm{fg}}(j,k,\mathit{val}) \ , \end{aligned}$$

where the function $\mathit{send}^{\mathrm{fg}} : I \times I \times \mathit{val} \rightarrow \mathit{MSG}$ models the send function for a faulty sending process ('fg' denotes *fault-generating*). The simplest means of modeling this is with an uninterpreted function defined in Def. 10. We provide no definition for $\mathit{bad\_send}^{\mathrm{fg}}$ – when applied to its arguments, it is an undefined constant in *MSG*. Particularly, $\mathit{bad\_send}^{\mathrm{fg}}(j,k)$ does not necessary equal $\mathit{bad\_send}^{\mathrm{fg}}(j,i)$, so this successfully models Byzantine faults. Other faults, such as symmetric or benign faults [29], are modeled this way, too.[7]

Modeling the Algorithm

We construct the fault-free specification first. To model the first round of communication in OM_1, we introduce the function $\mathit{update}_0^{\mathrm{ff}} : I \times \mathit{VAL} \rightarrow \mathit{VAL}$ modeling a process correctly sending its message to all other processes (the subscript denotes the round of communication modeled). We define $\mathit{update}_0^{\mathrm{ff}}$ as

$$\mathit{update}_0^{\mathrm{ff}}(j,\mathit{val}) \stackrel{\mathrm{df}}{=} \lambda k.\ \mathit{send}^{\mathrm{ff}}(j,k,\mathit{val}) \ .$$

The image of $\mathit{update}_0^{\mathrm{ff}}(j,\mathit{val})$ is a function that takes a receiver k and returns the value set to k by j.

To model the second round of communication, we introduce the function $\mathit{update}_1^{\mathrm{ff}} : \mathit{VAL} \rightarrow \mathit{VAL}$ that takes a process function and updates it by modeling each process sending its message to every other process and then each process computing the majority of the messages it has received. It is defined as

---

[6] For the purposes of this paper, we assume that all faults are permanent and ignore the effects of transient faults in which faulty nodes may work again at some point in the future.

[7] In this respect, the model represents a weakening of the algorithm. For example, if a sender fails-silent, the receiver has knowledge of this and relays a constant value. Representing this value with an uninterpreted constant belies this. Under a more refined fault model such as the Hybrid Fault Model [29], the kinds of faulty behaviors that senders exhibit are distinguished, and fail-silent messages would have a special representation.

$$\begin{aligned}&\mathit{update}_1^{\mathrm{ff}}(\mathit{val}) \overset{\mathrm{df}}{=}\\&\quad \lambda k.\, \mathit{maj}(\lambda j.\, \mathit{send}^{\mathrm{ff}}(j,k,\mathit{val}))\ .\end{aligned}$$

The function $\mathit{maj} : \mathit{VAL} \rightarrow \mathit{MSG}$ models majority voting. A majority vote returns the message that the majority of the processes contain, if one exists. Otherwise, a special constant is returned. The function maj can be defined constructively by a particular algorithm that computes the majority function (e.g., Boyer and Moore's very efficient `MJRTY` algorithm [4]), or the maj can be axiomatized so that it is implementation-independent [26]. For our purposes, we need only introduce one simple axiom, stated in Sect. 6.3.

Putting the two stages together, the fault-free specification of OM_1 is the recursive function $OM_1^{\mathrm{ff}} : (I \times \mathit{VAL} \times \{0,1\}) \rightarrow \mathit{VAL}$. It takes as arguments the general's process index, the current set of messages the processes contain, and the current round (either 0 or 1). It returns an updated process function modeling the output of OM_1 for each process. Let $\mathit{rnd} \in \{0,1\}$.

$$\begin{aligned}&OM_1^{\mathrm{ff}}(g,\mathit{val},\mathit{rnd}) \overset{\mathrm{df}}{=}\\&\quad \text{if } \mathit{rnd}=0\\&\quad \text{then } OM_1^{\mathrm{ff}}(g, \mathit{update}_0^{\mathrm{ff}}(g,\mathit{val}), \mathit{rnd}+1)\\&\quad \text{else } \mathit{update}_1^{\mathrm{ff}}(\mathit{val})\ .\end{aligned}$$

Although OM_1^{ff} takes a process function val as an argument, the only input of concern is that of g, the general.

Now, we provide the fault-tolerant specification. This specification is quite similar to the fault-free one. We need only to define analogous update functions allowing faults to occur. The following functions have the same signatures as their fault-free counterparts. Let $\mathit{update}_0^{\mathrm{fg}} : I \times \mathit{VAL} \rightarrow \mathit{VAL}$ be defined as

$$\mathit{update}_0^{\mathrm{fg}}(j,\mathit{val}) \overset{\mathrm{df}}{=} \lambda k.\, \mathit{send}^{\mathrm{fg}}(j,k,\mathit{val})\ ,$$

and let $\mathit{update}_1^{\mathrm{fg}} : \mathit{VAL} \rightarrow \mathit{VAL}$ be defined as

$$\begin{aligned}&\mathit{update}_1^{\mathrm{fg}}(\mathit{val}) \overset{\mathrm{df}}{=}\\&\quad \lambda k.\, \mathit{maj}(\lambda j.\, \mathit{send}^{\mathrm{fg}}(j,k,\mathit{val}))\ .\end{aligned}$$

We call the fault-tolerant specification OM_1^{ft} and define it as

$$\begin{aligned}&OM_1^{\mathrm{ft}}(g,\mathit{val},\mathit{rnd}) \overset{\mathrm{df}}{=}\\&\quad \text{if } \mathit{rnd}=0\\&\quad \text{then } OM_1^{\mathrm{ft}}(g, \mathit{update}_0^{\mathrm{fg}}(g,\mathit{val}), \mathit{rnd}+1)\\&\quad \text{else } \mathit{update}_1^{\mathrm{fg}}(\mathit{val})\ .\end{aligned}$$

Modeling the MFA

As mentioned, the MFA constrains the fault-generating environments. The formalization of MFA given in Def 13 is straightforward. We constrain the size of the sets of the non-faulty and faulty nodes, respectively.

$$\begin{aligned}&\mathit{MFA} \overset{\mathrm{df}}{=}\\&\quad |\{j \mid \mathit{good?}(j)\}| \geq 3 \text{ and}\\&\quad |\{j \mid \text{not } \mathit{good?}(j)\}| \leq 1\ .\end{aligned}$$

6.3 Verifying OM_1

Because we left the majority function uninterpreted, we need to state a small axiom about its behavior. Our axiom states that if more than half the processes contain the same message, then that message is the majority of all messages in the system.

Axiom 1 (Majority). *If there exists* $msg \in MSG$ *such that* $2 \times |\{j \mid val(j) = msg\}| > |\{j \mid \text{true}\}|$, *then* $maj(val) = msg$.

We now prove that the OM_1 algorithm satisfies Def. 9.

Theorem 2 (Fault-Tolerant Correctness). $OM_1^{\text{ft}}(g, val, 0)$ *satisfies fault-tolerant correctness.*

Proof. The input of $OM_1^{\text{ft}}(g, val, 0)$ and $OM_1^{\text{ff}}(g, val, 0)$ is $val(g)$, the message with which the general is initialized (for OM_1, the messages other processes are initialized with are irrelevant). We must show that for any value of $OM_1^{\text{ft}}(g, val, 0)$ allowed by the *MFA* and for any $l \in I$, if $good?(l)$, then there exists a function val' such that $val(g) = val'(g)$, and $OM_1^{\text{ft}}(g, val, 0)(l) = OM_1^{\text{ff}}(g, val', 0)(l)$.

1. First, we compute the value of $OM_1^{\text{ff}}(g, val, 0)$ by definition expansion and λ-application.
$$OM_1^{\text{ff}}(g, val, 0) = \lambda k.\, maj(\lambda j.\, val(g)) \ ,$$
and since every process contains the same message,
$$\lambda k.\, maj(\lambda j.\, val(g)) = \lambda k.\, val(g) \ ,$$
by Ax. 1. Note that $OM_1^{\text{ff}}(g, val, 0)$ computes the value of g for all processes k, as expected.

2. Now we produce a function val' such that $val(g) = val'(g)$, and for all $l \in I$ where $good?(l)$, $OM_1^{\text{ft}}(g, val, 0)(l) = OM_1^{\text{ff}}(g, val', 0)(l)$. From Step 1, it follows that
$$OM_1^{\text{ff}}(g, val', 0)(l) = val'(g) \ ,$$
by λ-application. By definition expansion and λ-application,
$$OM_1^{\text{ft}}(g, val, 0)(l) = maj(\lambda j.\, send^{\text{fg}}(j, l, \lambda i.\, send^{\text{fg}}(g, i, val))) \ .$$

3. We consider two cases.
   a) If $good?(g)$, then from Step 2, definition expansion, and λ-application,
   $$OM_1^{\text{ft}}(g, val, 0)(l) = maj\left(\lambda j. \begin{cases} \text{if } good?(j) \text{ then } val(g) \\ \text{else } send^{\text{fg}}(j, l, \lambda i.\, val(g)) \end{cases}\right) \ .$$
   By the *MFA*, at least three processes contain $val(g)$ and no more than one process does not. Thus,
   $$maj\left(\lambda j. \begin{cases} \text{if } good?(j) \text{ then } val(g) \\ \text{else } send^{\text{fg}}(j, l, \lambda i.\, val(g)) \end{cases}\right) = val(g) \ ,$$
   by Ax. 1.

b) If not $good?(g)$, val' can be any function such that $OM_1^{\mathrm{ft}}(g, val, 0)(l) = OM_1^{\mathrm{ff}}(g, val', 0)(l)$, so let

$$val' = \lambda g.\, maj(\lambda j.\, send^{\mathrm{fg}}(j, k, \lambda i.\, send^{\mathrm{fg}}(g, i, val)))\ .$$

The equality holds from Steps 1 and 2.

□

We should note that because we abstracted faults as the inability of a process to send messages, but allow them to receive and compute messages correctly, we did not need to assume that $good?(l)$ holds in Step 2. This is a consequence of our abstraction (that simplifies proofs). Nevertheless, for the actual algorithm, we are only concerned about the output of non-faulty processes.

The proofs that OM_1 satisfies agreement and validity are now trivial.

Theorem 3 (OM_1 Agreement). $OM_1^{\mathrm{ff}}(g, val, 0)$ *satisfies Agreement.*

Proof. From Step 1 of Thm. 2,

$$OM_1^{\mathrm{ff}}(g, val, 0)(i) = val(g) = OM_1^{\mathrm{ff}}(g, val, 0)(j)\ .$$

for processes i, $j \in I$. □

Theorem 4 (OM_1 Validity). $OM_1^{\mathrm{ff}}(g, val, 0)$ *satisfies Validity.*

Proof. Suppose $good?(g)$. Then for $i \in I$, $OM_1^{\mathrm{ff}}(g, val, 0)(i) = \lambda k.\, val(g)(i) = val(g)$ by Step 1 of Thm. 2. □

Theorem 5 (Correctness). *If MFA holds, then* $OM_1^{\mathrm{ft}}(g, val, rnd)$ *satisfies agreement and validity.*

Proof. Immediate from Thms. 1, 2, 3, and 4. □

7 Conclusion

We have provided a method by which proofs of correctness for fault-tolerant algorithms can be decomposed. This is in the framework of recursive function specifications. The method is demonstrated by specifying the Oral Messages(1) algorithm, and verifying its correctness.

Hard problems do not just go away. One of the hard problems in verifying fault-tolerant algorithms is reasoning about the nondeterminism of fault transitions. We have traded the problem of nondeterministic execution for a set of fixed inputs (in a fault-generating environment) to the problem of determining the appropriate inputs for deterministic execution (in a fault-free environment).

Still, we believe the decomposition presented here allows for simpler and more systematic proofs. By this decomposition, faults need be reasoned about explicitly only in demonstrating fault-tolerant correctness. If this holds, faults can be ignored in proving the algorithm satisfies its requirements. This is especially beneficial if the requirements are numerous.

A common challenge for new tools and techniques in formal methods is, "Does this method scale?" [10]. Admittedly, OM_1 is a simple algorithm, and there are relatively simple non-compositional proofs of its correctness and the correctness of the Oral Messages algorithm for an arbitrary number of faults [17].[8] It therefore may seem as if not much is gained through this verification technique. However, OM_1 is presented only for illustrative purposes. We believe (but have not yet experimentally verified) that proofs of correctness for complex algorithms stand to benefit even more from the technique.

Finally, a fault-free specification disentangles the specification of an algorithm and the specification of the environment in which it executes. A fault-free specification is one from which an implementation of the algorithm can possibly be derived. In fact, an implementation can be formally derived using a design algebra [3, 9]. This allows for potentially more formal connections between the fault-tolerant specification of an algorithm, about which correctness conditions are proved to hold, and its fault-free specification, which ignores the effects of the environment and is closer to a hardware or software implementation. The upshot is greater assurance of correct design for safety-critical systems.

Acknowledgments

Conversations with Steven D. Johnson, Paul S. Miner, and Ricky W. Butler motivated this work.

References

1. Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *International Conference on Distributed Computing Systems*, pages 436–443, 1998.
2. John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), 1978. 1977 ACM Turing Award Lecture.
3. Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Indiana University, December 1994.
4. Robert S. Boyer and J. Stother Moore. MJRTY—a fast majority voting algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1, pages 105–117. Kluwer Academic Publishers, 1991.
5. K. Mani Chandy and Jayadev Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.
6. Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), February 1991.
7. Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in

---

[8] This is not to say that all formal proofs of its correctness have been simple – see [15].

Computer Science, pages 235–248. The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP, Springer-Verlag Heidelberg, September 2003.

8. Alfons Geser and Paul Miner. A formal correctness proof of the SPIDER diagnosis protocol. Technical Report 2002-211736, NASA Langley Research Center, Hampton, Virginia, August 2002. Technical Report contains the Track B proceedings from Theorem Proving in Higher Order Logics (TPHOLSs).
9. Steven D. Johnson. *Synthesis of Digital Designs fom Recursion Equations.* ACM Distinguished Dissertations. MIT Press, 1983.
10. Steven D. Johnson. View from the fringe of the fringe. In Tiziana Margaria and Thomas Melham, editors, *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2001.
11. Steven D. Johnson. Formal methods in embedded design. *Computer*, pages 104–106, Novemeber 2003.
12. Hermann Koptez. *Real-Time Systems.* Kluwer Academic Publishers, 1997.
13. Lamport, Shostak, and Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.
14. Leslie Lamport. Composition: A way to make proofs harder. *Lecture Notes in Computer Science*, 1536:402–423, 1998.
15. Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In *FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS*, LNCS. Springer-Verlag, 1994.
16. Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc., 1981.
17. Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.
18. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
19. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* SRI International, version 2.4 edition, December 2001. Available at `http://pvs.csl.sri.com/manuals.html`.
20. Sam Owre, John Rusby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
21. M. Pease, R. Shostak, and L Lamport. Reaching agreement in the presence of faults. *Journal of of the ACM*, 27(2):228–234, 1980.
22. James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.
23. Holger Pfeifer. *Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture.* PhD thesis, Universität Ulm, 2003. Available at `http://www.informatik.uni-ulm.de/ki/Papers/pfeifer-phd.html`.
24. Lee Pike, Jeffery Maddalon, Paul Miner, and Alfons Geser. Abstractions for fault-tolerant distributed system verification. In *Proceedings of Theorem Proving in Higher-Order Logics (TPHOLs)*, 2004. Accepted. Available at `http://shemesh.larc.nasa.gov/fm/spider/spider_pubs.html`.

25. John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report Contractor Report 4551, NASA, December 1993.
26. John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, sep 1999.
27. John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001.
28. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4), December 1990.
29. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.
30. Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 85–97, Vienna, Australia, September 1992. Third IFIP International Working Conference on Dependable Computing for Critical Applications, Springer-Verlag.
31. Doug G. Weber. Fault tolerance as self-similarity. In Jan Vytopil, editor, *Formal Techniques in Real-Time Fault-Tolerant Systems*, pages 33–49. Kluwer Academic Publishers, 1993.
32. William D. Young. Comparing verification systems: Interactive consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, April 1997.